Premier makefile

Rappel préliminaire : l'excellent polycopié sur le langage C de H. Garetta est disponible à l'url `http://www.dil.univ-mrs.fr/~garreta/Polys/PolyC.pdf`

1 Objectif de ce document

Ce document a pour objectif de familiariser les étudiants avec l'utilisation des Makefiles, la programmation modulaire et la compilation séparée.

2 Application

L'application que nous allons développer est une application très simple qui consiste simplement à fournir la possibilité de faire des opérations de base : la multiplication, la division, la somme et le produit de deux chiffres. Evidemment, il n'est pas nécessaire d'écrire un programme, encore moins modulaire, pour effectuer ces opérations en C, mais c'est un exemple qui permet de montrer comment mettre en œuvre le principe de programmation modulaire.

2.1 Découpage en modules

La phase la plus importante de la programmation modulaire est d'identifier les modules à écrire. Ici, nous avons deux types d'opération : celles qui sont reliées à l'opérateur 'x', ie la multiplication et celles reliées à l'opérateur '+', ie l'addition. Les premières concernent la multiplication proprement dite et la division et les secondes l'addition elle-même et la soustraction.

Nous déciderons donc de définir deux modules : le premier prendra en charge les opérations de type 'x' et le second les opérations de type '+'. Nous appellerons le premier module `produit` et le second `somme`.

Le découpage fonctionnel en modules est maintenant établi. Il ne reste plus qu'à programmer les fonctions qui correspondent aux modules identifiés.

2.2 Programmation

L'écriture d'un module se fait généralement par la mise en place de deux fichiers. Un fichier en-tête ou *header* dont le nom est du type `nommodule.h` et un fichier de définition, où la programmation proprement dite est effectuée, qui est un fichier de type `nommodule.c`. Le fichier `stdio.h` dont vous avez l'habitude, est donc le fichier header du module qui concerne l'ensemble de toutes les fonctionnalités standard d'entrées/sorties à l'écran et dans les fichiers.

2.2.1 Fichiers header/en-tête

Le fichier header, est un fichier qui ne contient que les signatures, commentées, des fonctions qui constituent le module concerné. Eventuellement, il contient également la déclaration et la définition de types (en particulier des `struct` et des pointeurs sur `struct`). Un header doit être vu comme un « contrat » portant sur les fonctionnalités qui seront fournies par le module, une fois celui-ci programmé. Pour vous donner une idée de ce à quoi peut ressembler un header, vous pouvez par exemple regarder le fichier `stdio.h`, qui se trouve généralement dans l'arborescence de votre système de fichiers à l'emplacement `/usr/include/stdio.h` (attention de ne pas modifier ce fichier, et donc de ne pas l'ouvrir avec les droits administrateur).

Suivant la politique de découpage d'un module en deux fichiers, nous allons donc créer 4 fichiers pour notre petite application : `produit.h`, `produit.c`, `somme.h` et `somme.c`.

Le contenu des deux fichiers `produit.h` et `somme.h` est indiqué dans les figures 1 et 2.

```
#ifndef __PRODUIT__
#define __PRODUIT__

/** Calcule le produit entre les deux nombres reels passes en parametre    */
float multiplication(float, float);

/** Calcule la division du premier reel par le second. Quitte le programme */
/** si le second parametre est nul.                                        */
float division(float, float);

#endif
```

FIG. 1 – Le header `produit.h`.

```
#ifndef __SOMME__
#define __SOMME__

/** Calcule la somme entre les deux nombres reels passes en parametres */
float addition(float, float);

/** Calcule la difference entre le premier nombre et le second         */
float soustraction(float, float);

#endif
```

FIG. 2 – Le header `somme.h`.

Quelques commentaires sur ces fichiers. Comme on l'a dit précédemment, ils ne contiennent que les signatures des fonctions (type de retour, identificateur de la fonction, type des paramètres pris par les fonctions). Il s'agit donc du contrat que promettent de remplir les modules `produit` et `somme`. Dans le cadre d'un projet fait à plusieurs personnes, l'un des programmeur considèrera que les fonctions déclarées dans ces fichiers headers seront finalement programmées un jour et qu'il peut donc les utiliser dans différentes parties de son code (par le biais de la directive bien connue #include. Les macros #ifndef __XXX__, #define __XXX__ et #endif sont des *macros* qu'il est habituel de mettre dans les fichiers headers. Elles permettent d'éviter des inclusions circulaires de headers : on peut faire un #include d'un header dans un autre header, qui lui même peut faire un #include du premier fichier et l'inclusion des fichiers est sans fin si ces macros ne figurent pas dans les fichiers en-tête. L'utilisation des *underscores* (tirets '_') est également plus un usage qu'une obligation. Tout autre type d'identificateur C est accepté.

Comme on l'a dit, il nous est d'ores et déjà possible d'écrire un programme utilisant les fonctions déclarées dans les fichiers headers. Bien sûr, puisque la programmation des modules eux-mêmes (i.e. la *définition* des fonctions) n'a pas encore été faite, il n'est pas possible de réaliser une compilation complète du programme et donc d'obtenir un exécutable. Néanmoins, il est souvent utile (même si contre-intuitif d'un premier abord) de commencer éventuellement par écrire le programme principal, c'est-à-dire le programme contenant la fonction `main` le plus tôt possible. Cela permet notamment de bien identifier les fonctions utiles pour l'exécutable et elles seules. C'est un approche de programmation dite « descendante » (où on s'intéresse tout d'abord aux concepts de haut-niveau pour se concentrer ensuite aux aspects de techniques de la programmation).

La figure 3 propose un exemple très simple de programme principal utilisant les fonctions fournies par les modules. L'utilisation des fonctions déclarées dans les modules est possible grâce aux directives #include `"produit.h"` et #include `"somme.h"`. L'utilisation des doubles quotes " vient signifier que les header que le compilateur doit inclure ne sont pas des headers standard et que la recherche de ces fichiers

```c
#include <stdio.h>
#include "somme.h"
#include "produit.h"

int main(){
        float a, b;

        printf("Entrer un premier nombre reel : ");
        scanf("%f", &a);

        printf("Entrer un second nombre reel : ");
        scanf("%f", &b);

        printf("\n%f + %f = %f\n", a, b, addition(a, b));
        printf("%f - %f = %f\n", a, b, soustraction(a, b));
        printf("%f * %f = %f\n", a, b, multiplication(a, b));
        printf("%f / %f = %f\n", a, b, division(a, b));

        return 1;
}
```

FIG. 3 – Un programme principal `main.c` utilisant les fonctions fournies par les modules.

doit se faire dans le répertoire courant (l'utilisation de chemins absolus ou relatifs est également possible). Notons que pour la phase de *compilation* ce programme est correct, c'est-à-dire que syntaxiquement, il n'enfreint aucune règle du C et les fonctions utilisées "existent" puisque les headers le promettent. C'est pour l'*édition de liens* que la programmation en tant que telle des fonctions des modules est nécessaire.

2.2.2 Programmation des modules

La *définition* des fonctions, c'est-à-dire la programmation de ces fonctions afin qu'elles réalisent leur objectif, se fait dans les fichiers `monmondule.c` correspondant aux fichiers headers définis précédemment. Les figures 4 et 5 fournissent un exemple de programmation de ces modules.

Il n'y a pas grand chose à dire sur ces fichiers si ce n'est qu'ils incluent les fichiers headers qui leur correspondent (la compilation et l'édition de liens marchent sans cela, mais c'est une bonne habitude à prendre de faire ces inclusions, pour, notamment, la cohérence des fonctions déclarées et des fonctions

```c
#include "produit.h"

#include <assert.h>

float multiplication(float a, float b){
        return a*b;
}

float division(float a, float b){
        assert(b != 0); // regarder la valeur absolue serait mieux
        return a/b;
}
```

FIG. 4 – Le fichier de définition des fonctions `produit.c` du module `produit`.

```c
#include "somme.h"

float addition(float a, float b){
        return a+b;
}

float soustraction(float a, float b){
        return a-b;
}
```

FIG. 5 – Le fichier de définition des fonctions `somme.c` du module `somme`.

```
# Compilateur
CC=gcc -c -g -Wall

# Pour l'edition de liens
LD=gcc -o

all: main

main: main.o somme.o produit.o
        $(LD) main main.o somme.o produit.o

main.o: main.c somme.h produit.h
        $(CC) main.c

somme.o: somme.c somme.h
        $(CC) somme.c

produit.o: produit.c produit.h
        $(CC) produit.c
```

FIG. 6 – Un fichier Makefile pour notre petite application. Ce fichier peut également s'appeler makefile (avec un 'm' minuscule en début de nom).

définies). Autant que possible, il est préférable de mettre les #include nécessaires à la définition des fonctions plutôt dans le fichier `.c` que dans le `.h`. Il arrive cependant qu'il ne soit pas possible de faire autrement.

3 Compilation, édition de liens, makefiles

Il nous reste à présent à aborder la question de la compilation de ce projet en un seul programme à partir des multiples fichiers .c et .h. L'intérêt du fichier Makefile et de l'utilitaire make est précisément d'aider à cette tâche. En particulier, ces deux outils (bien que le seul programme soit en fait make, mais puisqu'il fonctionne avec un fichier qui s'appelle généralement Makefile, ce dernier est également considéré comme un outil), permettent de ne compiler que les parties qui l'exigent : si un module a été modifié depuis la dernière compilation complète du projet, alors make et le Makefile associé travailleront de concert pour ne recompiler que le module modifié et en intégrer les changements au programme principal. Plus généralement, un makefile peut être appréhender comme une recette de cuisine indiquant les ingrédients nécessaires à la création d'un exécutable et précisant comment les mélanger.

Un Makefile élémentaire pour le programme qui nous intéresse est donné dans la figure 6. Afin de créer l'exécutable correspondant à notre petite application, il suffit de taper la commande make all.

Une explication de texte rapide sur les Makefile. Les lignes importantes sont celles de la forme :

```
cible : dependances
    commande  # ligne précédée par une tabulation !!!
```

où il est possible d'avoir plusieurs commandes (sur plusieurs lignes). La cible est généralement l'objet que l'on veut obtenir (sauf pour la cible all), les dépendances sont les fichiers nécessaires à l'obtention de cette cible (les ingrédients) et la commande l'action à effectuer pour générer la cible à partir des dépendances (la méthode de cuisson). make permet de prendre en compte n'importe quelle cible et il serait possible de faire make produit.o : si le fichier produit.o existe déjà (comme provenant d'une compilation précédente) et que la date de dernière modification de produit.o est postérieure à celle de ses dépendances alors make n'exécute pas les commande permettant de générer produit.o et le fichier existant n'est pas modifié (la logique est que le fichier produit.o est à jour par rapport aux fichiers qui permettent de le construire) ; inversement si les dépendances ont une date de modification ultérieure à celle de la cible alors celle-ci est recompilée (l'ancien produit.o est alors remplacé par un nouveau produit.o).

Notons que deux types d'actions importantes doivent être relevés : la compilation et l'édition de liens. La compilation est l'opération qui produit un fichier objet `.o` à partir d'un fichier `.c` : il s'agit de la traduction d'un fichier C en langage machine, traduction qui repose sur la correction du langage utilisé (syntaxe correcte, fonctions utilisées déclarées dans des headers inclus dans le fichier `.c`). La phase de compilation ne donne pas d'exécutable car comme on peut le voir, la production des fichiers `.o` se fait module par module et il n'y pas vraiment de phase de « fusion » des différentes fonctionnalités. Cette mise en commun des fonctionnalités des modules se fait lors de la phase d'édition de liens, qui produit l'exécutable cible (ici, du nom de `main`), et qui vérifie que les fonctions dont la programmation était promise par les headers sont bel et bien programmées (et correctement programmées, évidemment). On rencontre souvent des erreurs lors de la création de l'exécutable : elles proviennent soit de la compilation soit de l'édition de liens. Si ces erreurs proviennent de la compilation, un message d'erreur assez compréhensible indique le fichier et l'endroit où se trouve l'erreur, qui est généralement une erreur de syntaxe. Si ces erreurs proviennent de la phase d'édition de liens, en revanche, le message d'erreur est alors un peu plus obscur, car il contient des adresses et des symboles autres que du texte simple. Le message d'erreur est néanmoins suffisamment explicite pour identifier facilement la cause de l'échec de l'édition de liens.

Pour ce projet, nous nous limiterons à l'écriture de fichiers Makefile assez sommaires et vous êtes donc invités à consulter le tutoriel très simple disponible à l'url suivante : `http://gl.developpez.com/tutoriel/outil/makefile/`. Tous les exemples donnés dans ce document fonctionnent normalement sans problème, vous pouvez vous initier aux makefile et à la compilation séparée à partir de ces quelques exemples.